Midiendo el tiempo de ejecución de un programa en Linux

Mario Medina (`mariomedina@udec.cl`)

1er. Semestre 2014

1. Introducción

Una premisa fundamental del análisis experimental de algoritmos es que es posible medir el tiempo de ejecución de un algoritmo en un computador dado.

El *tiempo de ejecución* de un programa es el tiempo "real" transcurrido entre dos eventos. Este tiempo es análogo a la medición entregada por un reloj o cronómetro, por lo que también se le conoce como *tiempo de reloj*.

El *tiempo de CPU* es el tiempo que la CPU ocupa en ejecutar el código de la aplicación. Generalmente se divide en el tiempo que la CPU pasa en modo usuario, y el tiempo que la CPU pasa en modo supervisor, ejecutando funciones del sistema operativo en beneficio del proceso. El comando `/usr/bin/time` puede ser invocado para obtener estos valores desde la línea de comando. Alternativamente, es posible utilizar diversas llamadas del sistema operativo como `time()` y `gettimeofday()` para obtener estos valores durante la ejecución del programa.

Existen muchas razones por las que estos dos tiempos no sean iguales: la ejecución concurrente de otros procesos y del sistema operativo, la ejecución simultánea de bloques de código en hardware paralelo, y el tiempo que el proceso dedica a esperar por operaciones de entrada/salida a recursos del sistema operativo o compartidos con otros procesos.

En las siguientes páginas, se detallan diferentes funciones disponibles para medir estos tiempos, mostrando fragmentos de código que ilustran su uso en lenguaje C.

Cabe hacer notar que el kernel del sistema operativo mantiene un reloj via software que cuenta *jiffies*, que puede ser 10, 4 ó 1 *ms*, dependiendo del valor de la variable de configuración del kernel `HZ`. Este valor limita la resolución de funciones como `getrusage()` y otras.

Para mayor información, el comando `man` permite acceder a la documentación estándar de estas funciones. Si las páginas de manual en español estén instaladas, entonces se recomienda usar el comando `man -L es`.

2. Midiendo el tiempo de ejecución

2.1. La función `time()`

La función `time()` retorna el número de segundos transcurridos desde el 1ro. de enero de 1970[1], y puede ser usada para medir el tiempo transcurrido durante la ejecución de un programa. La función retorna -1 en caso de error.

`time()`

```
#include <stdio.h>
#include <time.h>

int main(void)
{
   time_t startTime, endTime;
   unsigned int seconds;

   startTime = time(NULL);

   // CODE GOES HERE

   endTime = time(NULL);

   seconds = end - start;
   printf("Elapsed time (s): %d\n", seconds);
}
```

2.2. La función `gettimeofday()`

La función `gettimeofday()` retorna el tiempo transcurrido desde el 1ro. de enero de 1970 en una estructura `struct timeval` de dos miembros, `tv_sec` y `tv_usec`, que contienen el número de segundos y microsegundos transcurridos, respectivamente.

`struct timeval`

```
struct timeval {
   time_t   tv_sec;     // seconds
   suseconds_t tv_usec; // microseconds
};
```

La función `gettimeofday()` retorna 0 en caso de éxito, y -1 en caso de error.

Esta función puede ser usada para medir el tiempo transcurrido durante la ejecución de un programa, como se muestra en el siguiente código.

[1] El día 1ro. de enero de 1970, a las 00:00:00 UTC *(Universal Standard Time)*, es usado como referencia para muchas funciones y se conoce como el *Hito*, o *Epoch*.

gettimeofday()

```
#include <stdio.h>
#include <sys/time.h>

int main(void)
{
   struct timeval startTime, endTime;
   unsigned int micros;

   gettimeofday(&startTime);

   // CODE GOES HERE

   gettimeofday(&endTime);

   micros = (endTime.tv_sec - startTime.tv_sec)*1000000 + (endTime.tv_usec - startTime.tv_usec);

   printf("Elapsed time (us): %u\n", micros);
}
```

2.3. La función `clock_gettime()`

La función `clock_gettime()` puede ser usada para obtener el tiempo transcurrido desde el *Hito*. El primer argumento de la función indica el temporizador a utilizar. Valores posibles son `CLOCK_REALTIME`, un reloj de tiempo real de nivel de sistema, que puede ser usado para medir el tiempo transcurrido en la ejecución de un programa; el temporizador `CLOCK_MONOTONIC`, un reloj monotónico de nivel de sistema; `CLOCK_PROCESS_CPUTIME_ID`, un temporizador de nivel de proceso de alta resolución, que puede ser usado para medir el tiempo de CPU de un programa, y `CLOCK_THREAD_CPUTIME_ID`, un temporizador de nivel de hebra de alta resolución, que puede ser usado para medir el tiempo de CPU de la hebra en ejecución.

La función `clock_gettime()` recibe como segundo argumento una estructura `timespec`, que se muestra a continuación.

struct timespec

```
struct timespec {
   time_t tv_sec; // seconds
   long tv_nsec;  // nanoseconds
};
```

La función `clock_gettime()` también puede ser usada para obtener el tiempo de CPU de otro proceso o hebra, entregándole como primer argumento un identificador de temporizador, el cual a su vez puede obtenerse mediante las funciones `clock_getcpuclockid()` y `pthread_getcpuclockid()`.

Las funciones `clock_getcpuclockid()` y `pthread_getcpuclockid()` reciben como primer argumento un identificador de proceso (`pid`) o un identificador de hebra (`pthread`), y como segundo argumento la dirección de una variable de tipo `clockid_t`. Después de su ejecución, esta variable contiene el identificador del temporizador asociado al proceso o hebra respectiva, que se usará luego como argumento a `clock_gettime()`.

Invocar a `clock_getcpuclockid()` con el primer argumento `pid` igual a 0 retorna el identificador del temporizador del proceso en ejecución, por lo que su uso será entonces equivalente a usar la constante `CLOCK_PROCESS_CPUTIME_ID`. Asimismo, iInvocar a `pthread_getcpuclockid()` con el primer argumento `pthread` igual a `pthread_self()` retorna el identificador del temporizador de la hebra en ejecución, por lo que su uso será entonces equivalente a usar la constante `CLOCK_THREAD_CPUTIME_ID`.

Finalmente, la función `clock_getres()` retorna la resolución del temporizador utilizado en nanosegundos en una estructura `timespec`. Esta resolución depende de la implementación y no puede ser modificada.

Todas las funciones anteriores retornan 0 en caso de éxito, y -1 en caso de error.

Para utilizar estas funciones, es necesario enlazar el código con la biblioteca de tiempo real de Linux usando la opción `-lrt`. En el caso de usar la función `pthread_getcpuclockid()`, es necesario además enlazar el código con la opción `-lpthread`.

Finalmente, estas funciones no están definidas en el estándar ANSI, sino son funciones POSIX. Por ello, es necesario definir la constante simbólica `_POSIX_C_SOURCE` debe ser `199309L` ó `200112L`. La información contenida en las páginas de manual de estas funciones entrega más información al respecto.

clock_gettime()

```
#define _POSIX_C_SOURCE 199309L     // Necessary for clock_gettime()
//#define _POSIX_C_SOURCE 200112L     // Necessary for clock_getcpuclockid()
#include <stdio.h>
#include <sys/times.h>
#include <unistd.h>

int main(void)
{
   struct timespec start, end, resolution;
   unsigned long int nanos, res;

   clock_gettime(CLOCK_MONOTONIC, &start);

   // CODE GOES HERE

   clock_gettime(CLOCK_MONOTONIC, &end);

   clock_getres(CLOCK_MONOTONIC, &resolution);

   res = resolution.tv_sec*1000000000 + resolution.tv_nsec;
   printf("Resolution: %lu (ns)\n", res);

   nanos = (end.tv_sec - start.tv_sec)*1000000000 + end.tv_nsec - start.tv_nsec;
   printf("Elapsed time: %lu (ns)\n", nanos);
}
```

3. Midiendo el tiempo de CPU

3.1. La función `clock()`

La función `clock()` retorna una aproximación al tiempo de CPU utilizado por el programa hasta ese momento, en unidades de *clocks*. Este valor puede convertirse a segundos de CPU dividiendo el resultado por la constante del sistema `CLOCKS_PER_SEC`. Por su parte, el estándar POSIX requiere que esta constante siempre valga 1000000. Esto no significa que `clock()` tenga una resolución de microsegundos, sólo que los valores se reportan en esta unidad.

Cabe hacer notar que en Linux el tiempo de CPU retornado por `clock()` no incluye el tiempo de CPU de los procesos hijos.

Su uso se ilustra en el siguiente fragmento de código.

clock()

```
#include <stdio.h>
#include <time.h>

int main(void)
{
   clock_t start, end;
   unsigned long int micros;

   start = clock();
   // CODE GOES HERE
   end = clock();

   micros = end - start;
   printf("Clocks per sec: %lu\n", CLOCKS_PER_SEC);
   printf("CPU time (us): %lu\n", (micros*1000000)/CLOCKS_PER_SEC);
}
```

3.2. La función `times()`

La función `times()` retorna el tiempo de CPU utilizado por el proceso hasta ese momento, en una estructura `tms`. Los campos `tms_utime` y `tms_stime` contienen los tiempos de usuario y de sistema usados por el proceso hasta ese momento, mientras que los campos `tms_cutime` y `tms_cstime` contienen los tiempos de usuario y de sistema usados por sus procesos hijos hasta ese momento.

struct tms

```
struct tms {
   clock_t tms_utime;  // user time
   clock_t tms_stime;  // system time
   clock_t tms_cutime; // user time of children
   clock_t tms_cstime; // system time of children
};
```

Todos los valores retornados están en unidades de *ticks* de reloj. El número de ticks de reloj por segundo puede obtenerse mediante la llamada `sysconf(_SC_CLK_TCK)`.

Además, la función `times()` retorna el tiempo transcurrido desde un momento arbitrario en el pasado, por lo que puede ser utilizada para medir el tiempo de ejecución de un programa.

`times()`

```
#include <stdio.h>
#include <sys/times.h>
#include <unistd.h>

int main(void)
{
   clock_t startTime, endTime;
   struct tms startTms, endTms;
   unsigned int micros;
   long int ticksPerSec;

   startTime = times(&startTms);

   // CODE GOES HERE

   endTime = times(&endTms);

   micros = endTime - startTime;

   ticksPerSec = sysconf(_SC_CLK_TCK);

   printf("CLK_TCK = %ld\n", ticksPerSec);
   printf("Elapsed time (us): %lu\n", micros*1000000/ticksPerSec);
   printf("CPU user time (us): %lu\n",
      (endTms.tms_utime - startTms.tms_utime)*1000000/ticksPerSec);
   printf("CPU system time (us): %lu\n",
      (endTms.tms_stime - startTms.tms_stime)*1000000/ticksPerSec);
}
```

3.3. La función `getrusage()`

La función `getrusage()` puede ser usada para obtener información sobre la utilización de recursos del sistema consumidos por el proceso invocador, la hebra invocadora o los descendientes de éstos. Esta función rellena los elementos de una estructura `rusage`, la que se muestra a continuación.

Si el primer argumento de la función es la constante `RUSAGE_SELF`, esta estructura contiene información correspondiente al proceso invocador. En cambio, si el primer argumento es la constante `RUSAGE_CHILDREN`, esta estructura contiene información correspondiente a todos los descendientes del proceso invocador. Finalmente, si el primer argumento es la constante `RUSAGE_THREAD`, esta estructura contiene información correspondiente sólo a la hebra invocadora. Sin embargo, esta última opción es una extensión de GNU que sólo está disponible si se define la constante simbólica `_GNU_SOURCE`.

struct rusage

```
struct rusage {
   struct timeval ru_utime;   // user CPU time used
   struct timeval ru_stime;   // system CPU time used
   long ru_maxrss;            // maximum resident set size
   long ru_ixrss;             // integral shared memory size
   long ru_idrss;             // integral unshared data size
   long ru_isrss;             // integral unshared stack size
   long ru_minflt;            // page reclaims (soft page faults)
   long ru_majflt;            // page faults (hard page faults)
   long ru_nswap;             // swaps
   long ru_inblock;           // block input operations
   long ru_oublock;           // block output operations
   long ru_msgsnd;            // IPC messages sent
   long ru_msgrcv;            // IPC messages received
   long ru_nsignals;          // signals received
   long ru_nvcsw;             // voluntary context switches
   long ru_nivcsw;            // involuntary context switches
};
```

La función `getrusage()` retorna 0 en caso de éxito, y -1 en caso de error.

El código siguiente muestra cómo utilizar esta función para medir el tiempo de CPU ocupado por el proceso tanto en modo usuario como supervisor.

getrusage()

```
//#define _GNU_SOURCE      // Needed for RUSAGE_THREAD
#include <stdio.h>
#include <sys/resource.h>

int main(void)
{
   struct rusage start, end;
   long int utime, stime;

   getrusage(RUSAGE_SELF, &start);

   // CODE GOES HERE

   getrusage(RUSAGE_SELF, &end);

   utime = (end.ru_utime.tv_sec - start.ru_utime.tv_sec)*1000000 + end.ru_utime.tv_usec -
      start.ru_utime.tv_usec;
   stime = (end.ru_stime.tv_sec - start.ru_stime.tv_sec)*1000000 + end.ru_stime.tv_usec -
      start.ru_stime.tv_usec;
   printf("CPU user time: %lu microsec\n", utime);
   printf("CPU system time: %lu microsec\n", stime);
}
```

Cabe recordar que la resolución de la función `getrusage()` está limitada por la resolución del reloj del sistema, que por omisión tiene una resolución de 4 *ms*.

4. Resultados

Estas funciones se usaron para medir el tiempo que toma transponer una matriz de 4000×4000 números de doble precisión. Se escogió esta operación porque su simplicidad: se realiza íntegramente en memoria y no requiere operaciones de entrada/salida. La tabla 1 muestra el promedio y la desviación estándar de realizar 5 operaciones de transposición con las funciones descritas anteriormente, excepto la función `time()`, debido a su baja resolución.

Tabla 1: Tiempo para transponer una matriz de 4000×4000

Función	$\bar{x}$	σ
`clock()`	272000	4472
`times()`	266000	5477
`getrusage()`	267217	3347
`gettimeofday()`	274861	13539
`clock_gettime()`	267115	1705

En base a estos resultados, podemos afirmar que, si no es necesario una precisión del orden de nanosegundos, entonces se recomienda el uso de la función `times()`, que permite obtener simultáneamente el tiempo transcurrido y el tiempo de CPU de un programa.